邂逅JavaScript高级语法

王红元 coderwhy

课前提醒

- 上课时间：
  - 一般情况下，周一三五的晚上 20:00 ~ 22:00；
  - 如果因为放假或者特殊情况修改了时间，会在群里通知大家；
  - 建议来听直播，直播会更有感觉，一般当天上完课我就会上传录播；
- 上课的环境：
  - 上课我用的是Mac电脑，但是所有的操作和Windows电脑上是一致的（有不一致的我会专门录制视频）；
  - 上课我会使用VSCode作为开发工具，你也可以选择webstorm等其他工具；
- 相互尊重、共同进步：
  - 每个同学的基础不一样，之前的学习经历和方向不同；
  - 所讲的内容是为大部分同学考虑的，希望大家可以相互理解、相互帮助、共同进步；
- 课程资料：每次上完课，会将资料上传到腾讯课堂。

前端需要掌握的三大技术

- 前端开发最主要需要掌握的是三个知识点：HTML、CSS、JavaScript

JavaScript的重要性

JavaScript是前端万丈高楼的根基

前端行业在近几年快速发展，并且开发模式、框架越来越丰富。
但是不管你学习的是Vue、React、Angular，包括jQuery，以及一些新出的框架。
他们本身都是基于JavaScript的，使用他们的过程中你都必须好好掌握JavaScript。
所以JavaScript是我们前端万丈高楼的根基，无论是前端发展的万丈高楼，还是我们筑建自己的万丈高楼。

JavaScript在工作中至关重要

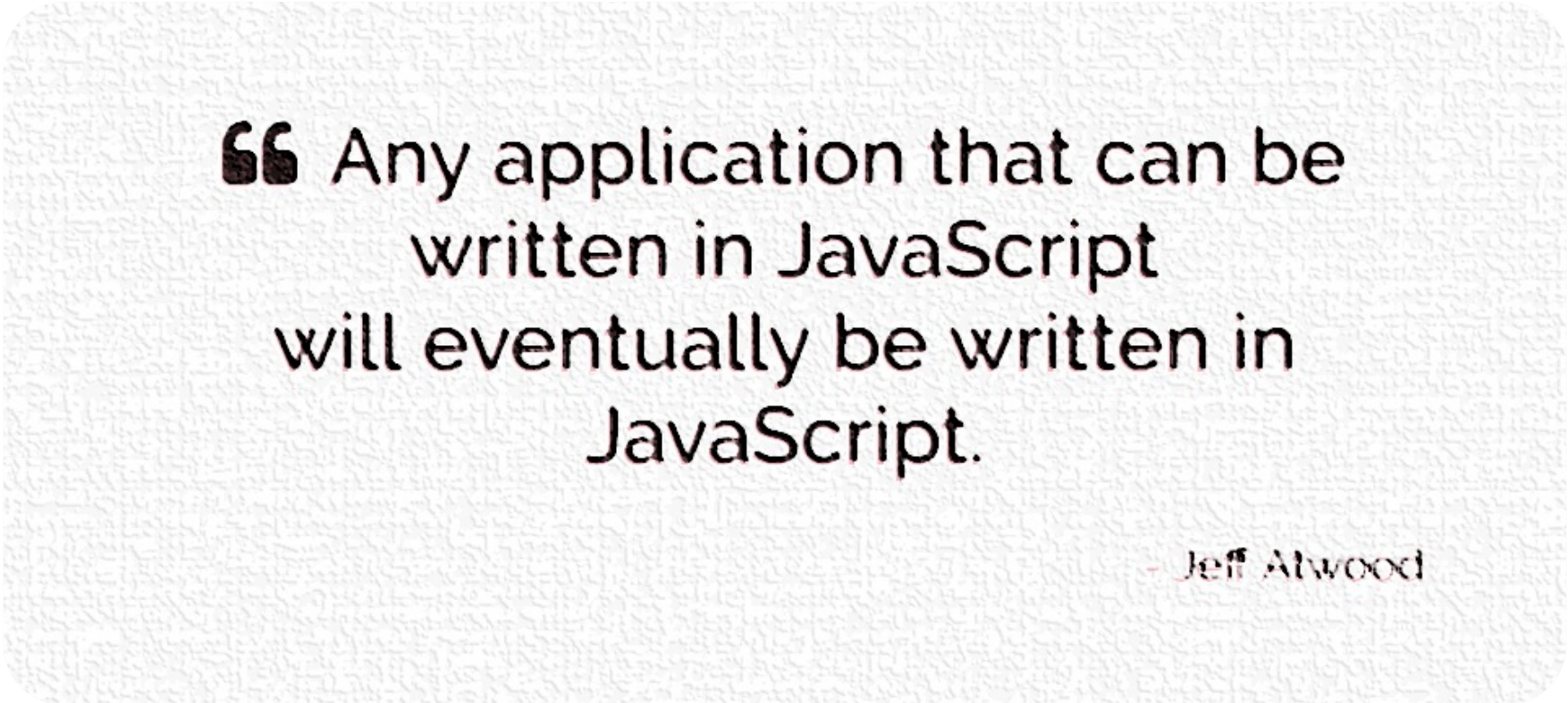
在工作中无论你使用什么样的技术，比如Vue、React、Angular、uniapp、taro、ReactNative。
也无论你做什么平台的应用程序，比如pc web、移动端web、小程序、公众号、移动端App。
它们都离不开JavaScript，并且深入掌握JavaScript不仅可以提高我们的开发效率，也可以帮助我们快速解决在开发中遇到的各种问题。
所以往往在面试时（特别是高级岗位），往往会考察更多面试者的JavaScript功底。

前端的未来依然是JavaScript

在可预见的前端的未来中，我们依然是离不开JavaScript的。
目前前端快速发展，无论是框架还是构建工具，都像雨后春笋一样，琳琅满目。
而且框架也会进行不断的更新，比如vue3、react18、vite2、TypeScript4.x。
前端开发者面对这些不断变化的内容，往往内心会有很多的焦虑，但是其实只要我们深入掌握了JavaScript，这些框架或者工具都是离不开JavaScript的

著名的Atwood定律

- Stack Overflow的创立者之一的 Jeff Atwood 在2007年提出了著名的 Atwood定律：
  - Any application that can be written in JavaScript, will eventually be written in JavaScript.
  - 任何可以使用JavaScript来实现的应用都最终都会使用JavaScript实现。



JavaScript应用越来越广泛

Web开发

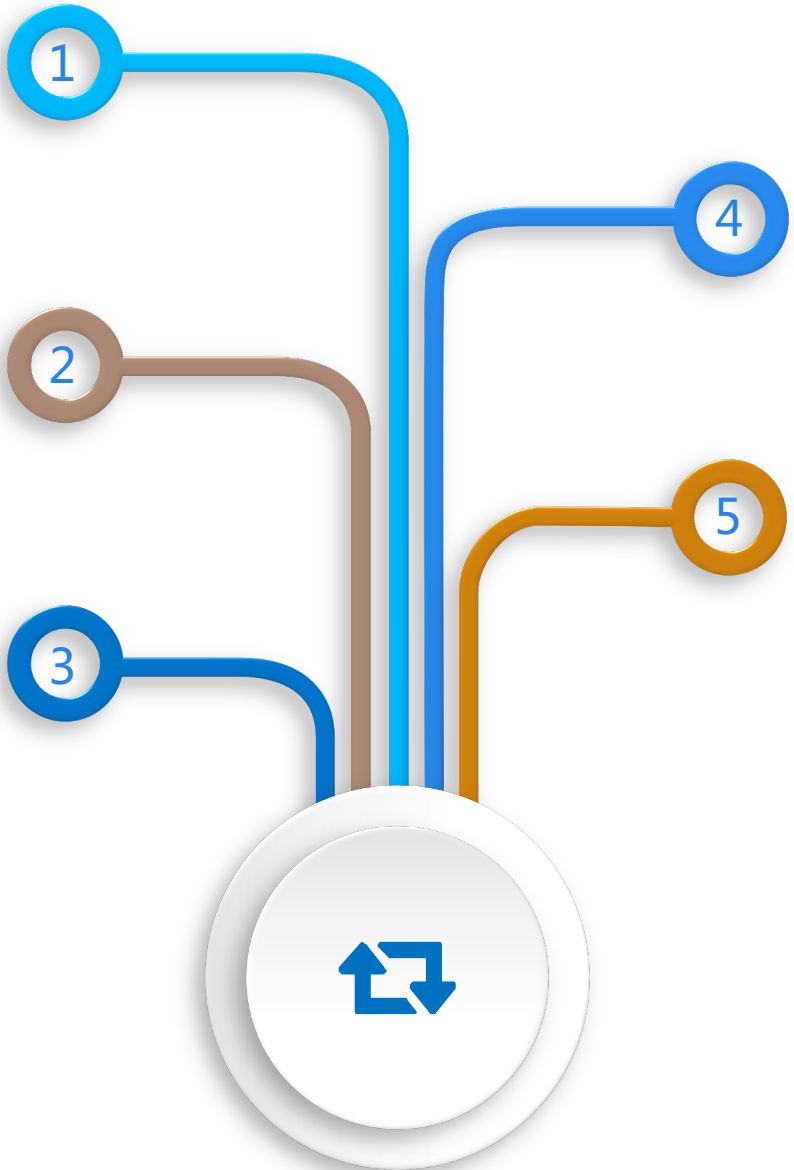
原生JavaScript
React开发
Vue开发
Angular开发

移动端开发

ReactNative
Weex

小程序端开发

微信小程序
支付宝小程序
uniapp
taro



桌面应用开发

Electron
比如VSCode

后端开发

Node环境
比如express、koa、egg.js

JavaScript让人迷惑的知识点

面向对象

JavaScript面向对象、继承、
原型、原型链等

函数、闭包

闭包的访问规则
闭包的内存泄露
函数中this的指向

ES新特性

ES6、7、8、9、10、11、12
新特性

作用域

作用域的理解、作用域提升、
块级作用域、作用域链、
AO、GO、VO等概念

其他一系列知识

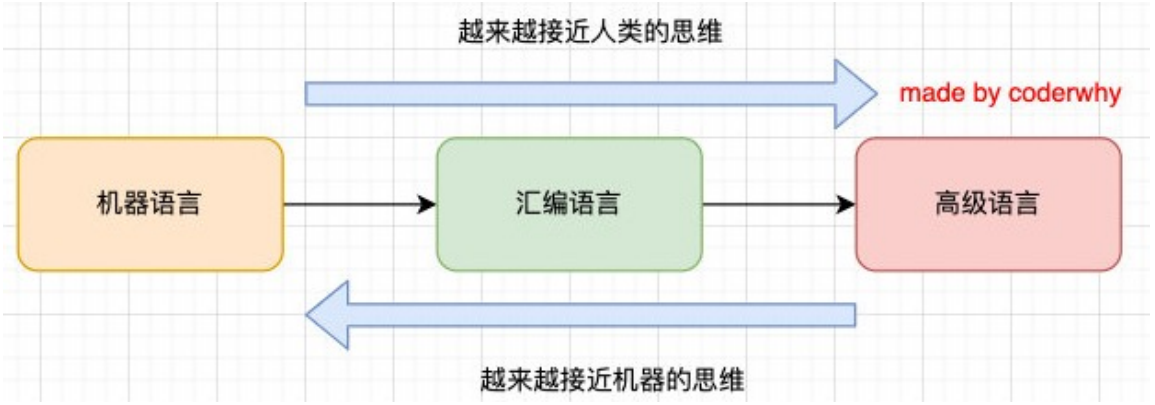
事件循环、微任务、宏任务、
内存管理、Promise、await、
asnyc、防抖、节流等等

TypeScript会取代JavaScript吗？

- TypeScript只是给JavaScript带来了类型的思维？
  - 因为JavaScript本身长期是没有对变量、函数参数等类型进行限制的；
  - 这可能给我们的项目带来某种安全的隐患；
- **在之后的JavaScript社区中出现了一系列的类型约束方案：**
  - 2014年，Facebook推出了flow来对JavaScript进行类型检查；
  - 同年，Microsoft微软也推出了TypeScript1.0版本；
  - 他们都致力于为JavaScript提供类型检查，而不是取代JavaScript；
- 并且在TypeScript的官方文档有这么一句话：源于JavaScript，归于JavaScript！
  - TypeScript只是JavaScript的一个超级，在它的基础之上进行了扩展；
  - 并且最终TypeScript还是需要转换成JavaScript代码才能真正运行的；
- 当然我们不排除有一天JavaScript语言本身会加入类型检测，那么无论是TypeScript，还是Flow都会退出历史舞台。

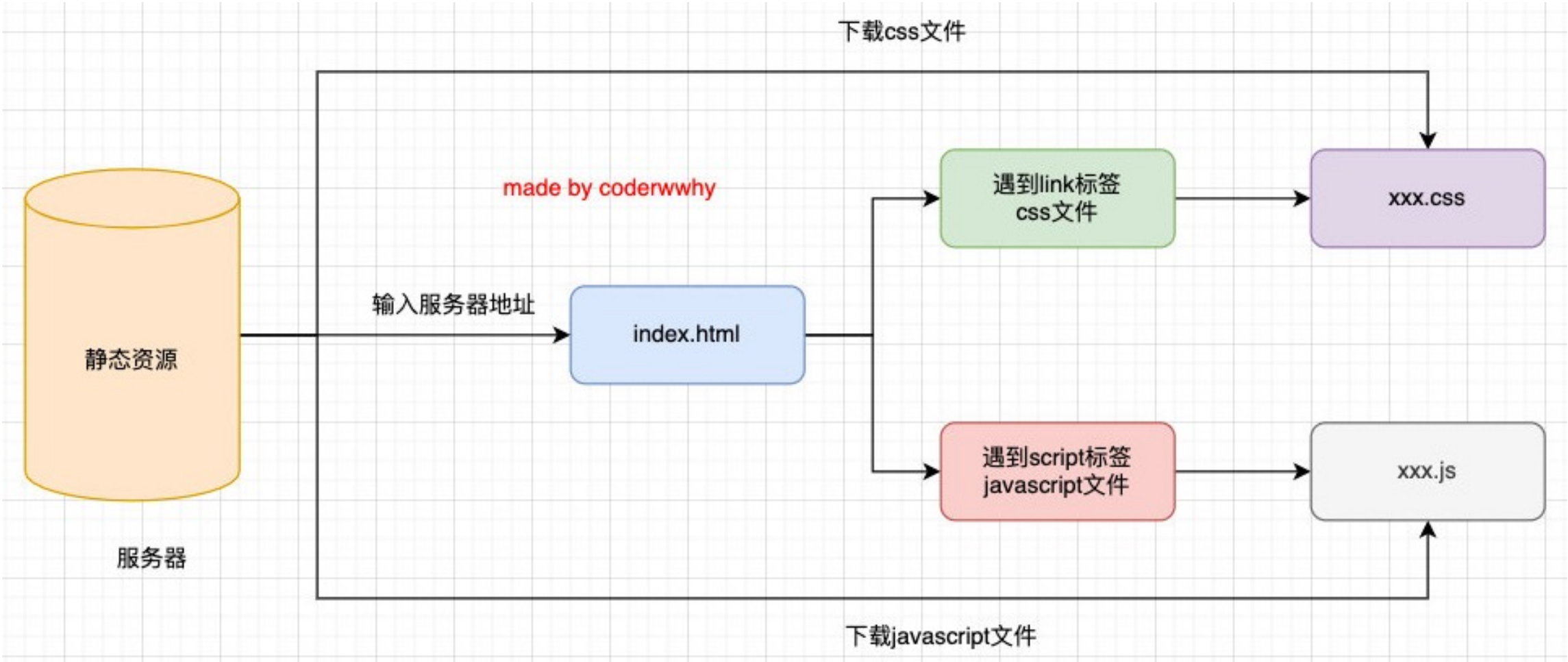
JavaScript是一门编程语言

- 为什么这里我要强调JavaScript是一门编程语言呢？很多同学想，我还不知道JavaScript是一门编程语言吗？
  - 事实上我们可以使用更加准备的描述是这样：**JavaScript是一门高级的编程语言。**
- 那么有高级编程语言，就有低级编程语言，从编程语言发展历史来说，可以划分为三个阶段：
  - 机器语言：1000100111011000，一些机器指令；
  - 汇编语言：mov ax,bx，一些汇编指令；
  - 高级语言：C、C++、Java、JavaScript、Python；
- 但是计算机它本身是不认识这些高级语言的，所以我们的代码最终还是需要被转换成机器指令：



浏览器的工作原理

■ 大家有没有深入思考过：JavaScript代码，在浏览器中是如何被执行的？

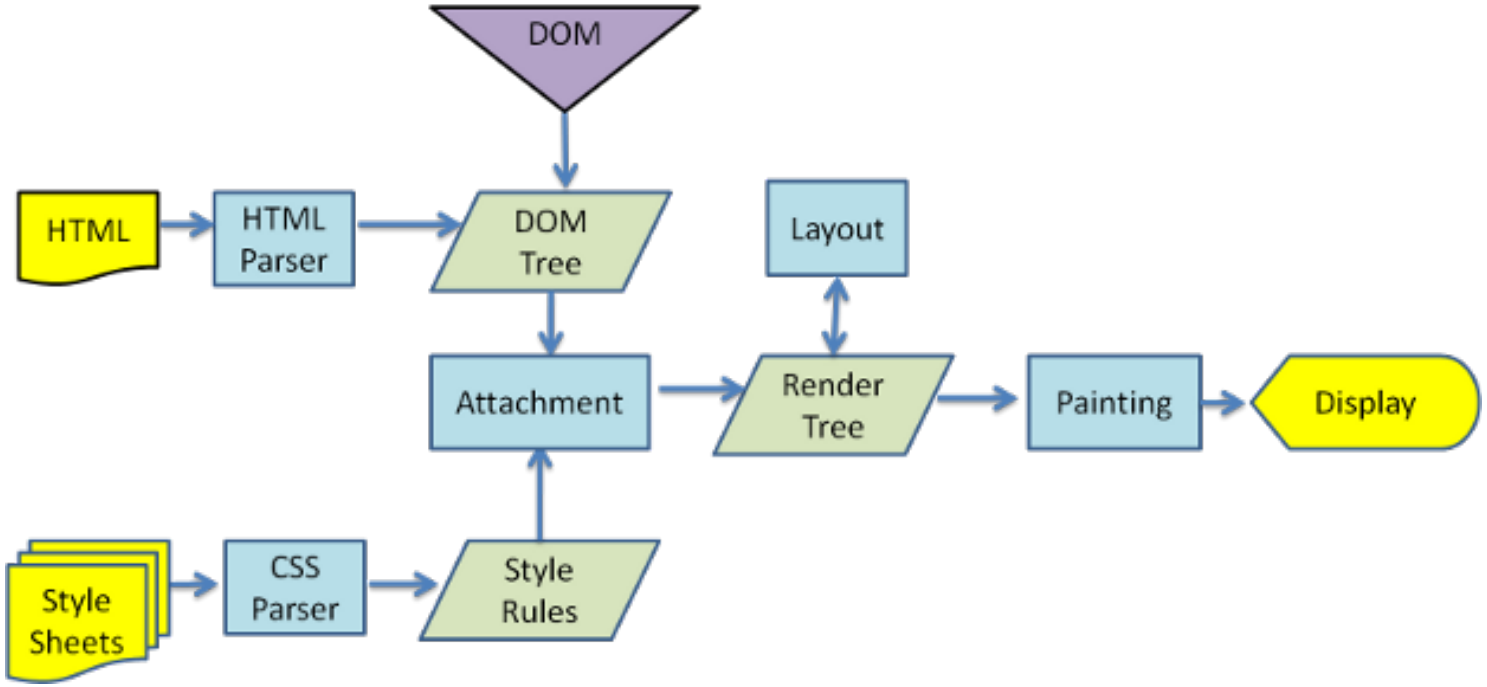


认识浏览器的内核

- 我们经常会说：不同的浏览器有不同的内核组成
  - Gecko：早期被Netscape和Mozilla Firefox浏览器浏览器使用；
  - Trident：微软开发，被IE4~IE11浏览器使用，但是Edge浏览器已经转向Blink；
  - Webkit：苹果基于KHTML开发、开源的，用于Safari，Google Chrome之前也在使用；
  - Blink：是Webkit的一个分支，Google开发，目前应用于Google Chrome、Edge、Opera等；
  - 等等...
- 事实上，我们经常说的浏览器内核指的是浏览器的排版引擎：
  - **排版引擎**（layout engine），也称为**浏览器引擎**（browser engine）、**页面渲染引擎**（rendering engine）或**样版引擎**。

浏览器渲染过程

- 但是在这个执行过程中，HTML解析的时候遇到了JavaScript标签，应该怎么办呢？
  - 会停止解析HTML，而去加载和执行JavaScript代码；



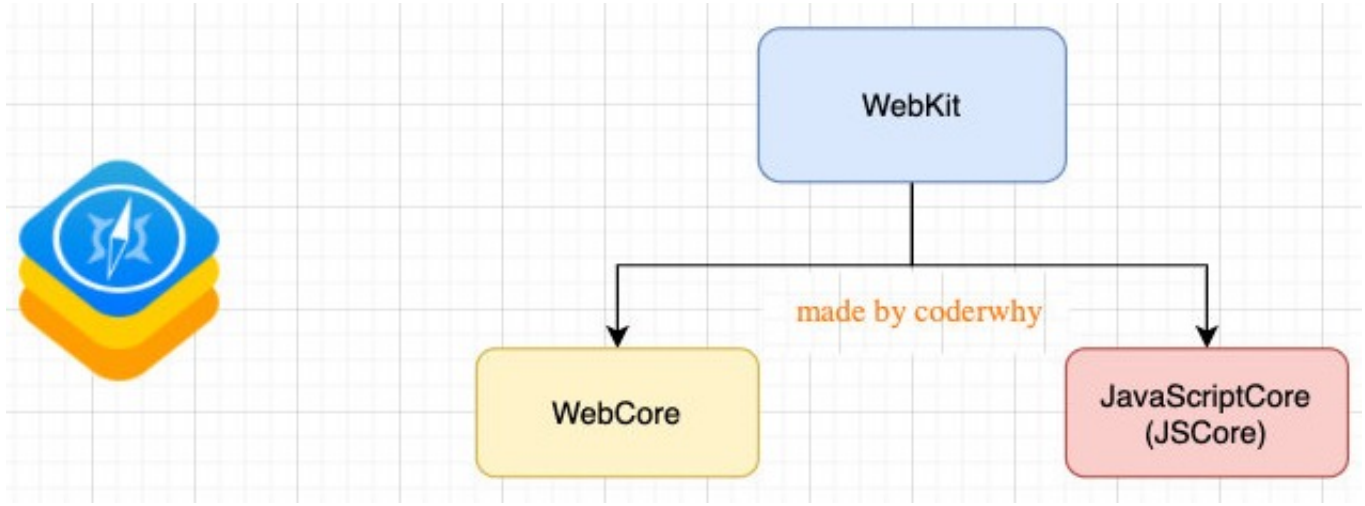
- 那么，JavaScript代码由谁来执行呢？
  - **JavaScript引擎**

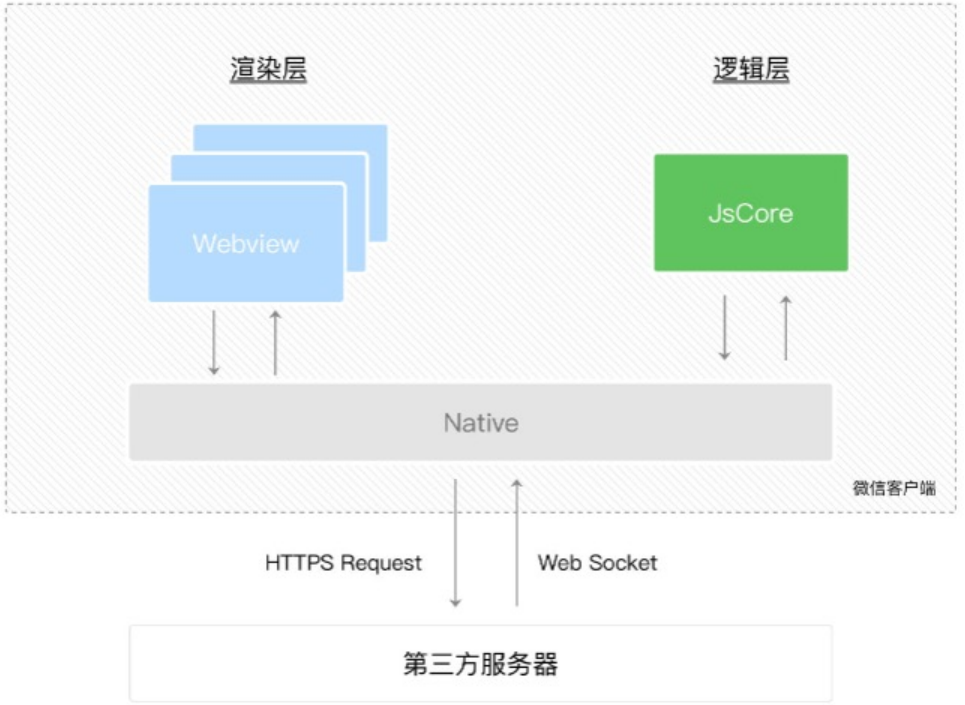
认识JavaScript引擎

- **为什么需要JavaScript引擎呢？**
  - 我们前面说过，高级的编程语言都是需要转成最终的机器指令来执行的；
  - 事实上我们编写的JavaScript无论你交给浏览器或者Node执行，最后都是需要被CPU执行的；
  - 但是CPU只认识自己的指令集，实际上是机器语言，才能被CPU所执行；
  - 所以我们需要JavaScript引擎帮助我们将JavaScript代码翻译成CPU指令来执行；
- **比较常见的JavaScript引擎有哪些呢？**
  - **SpiderMonkey**：第一款JavaScript引擎，由Brendan Eich开发（也就是JavaScript作者）；
  - **Chakra**：微软开发，用于IT浏览器；
  - **JavaScriptCore**：WebKit中的JavaScript引擎，Apple公司开发；
  - **V8**：Google开发的强大JavaScript引擎，也帮助Chrome从众多浏览器中脱颖而出；
  - 等等...

浏览器内核和JS引擎的关系

- 这里我们先以WebKit为例，WebKit事实上由两部分组成的：
  - **WebCore**：负责HTML解析、布局、渲染等等相关的工作；
  - **JavaScriptCore**：解析、执行JavaScript代码；
- 看到这里，学过小程序的同学有没有感觉非常的熟悉呢？
  - 在小程序中编写的JavaScript代码就是被JSCore执行的；

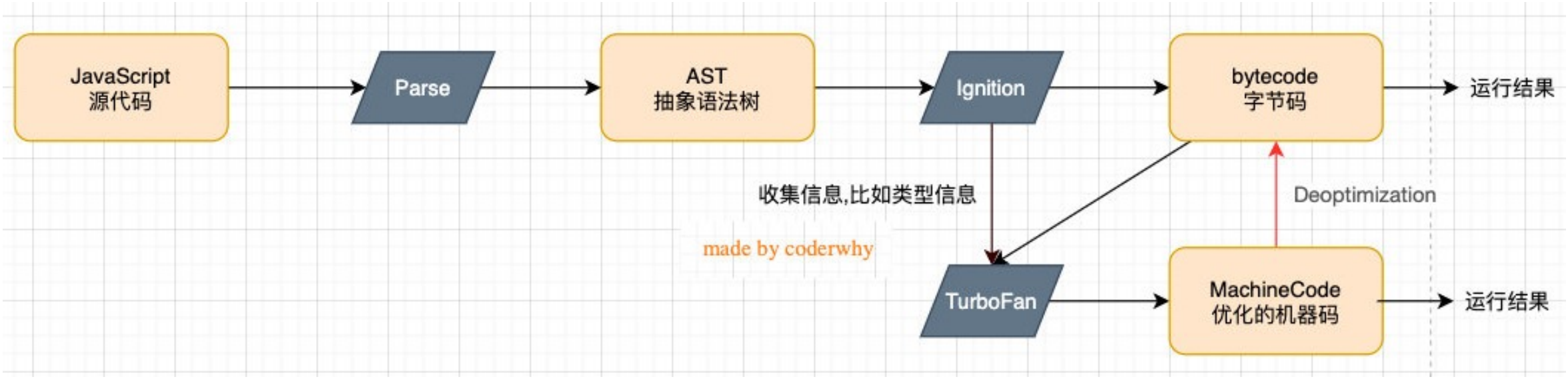




- 另外一个强大的JavaScript引擎就是V8引擎。

V8引擎的原理

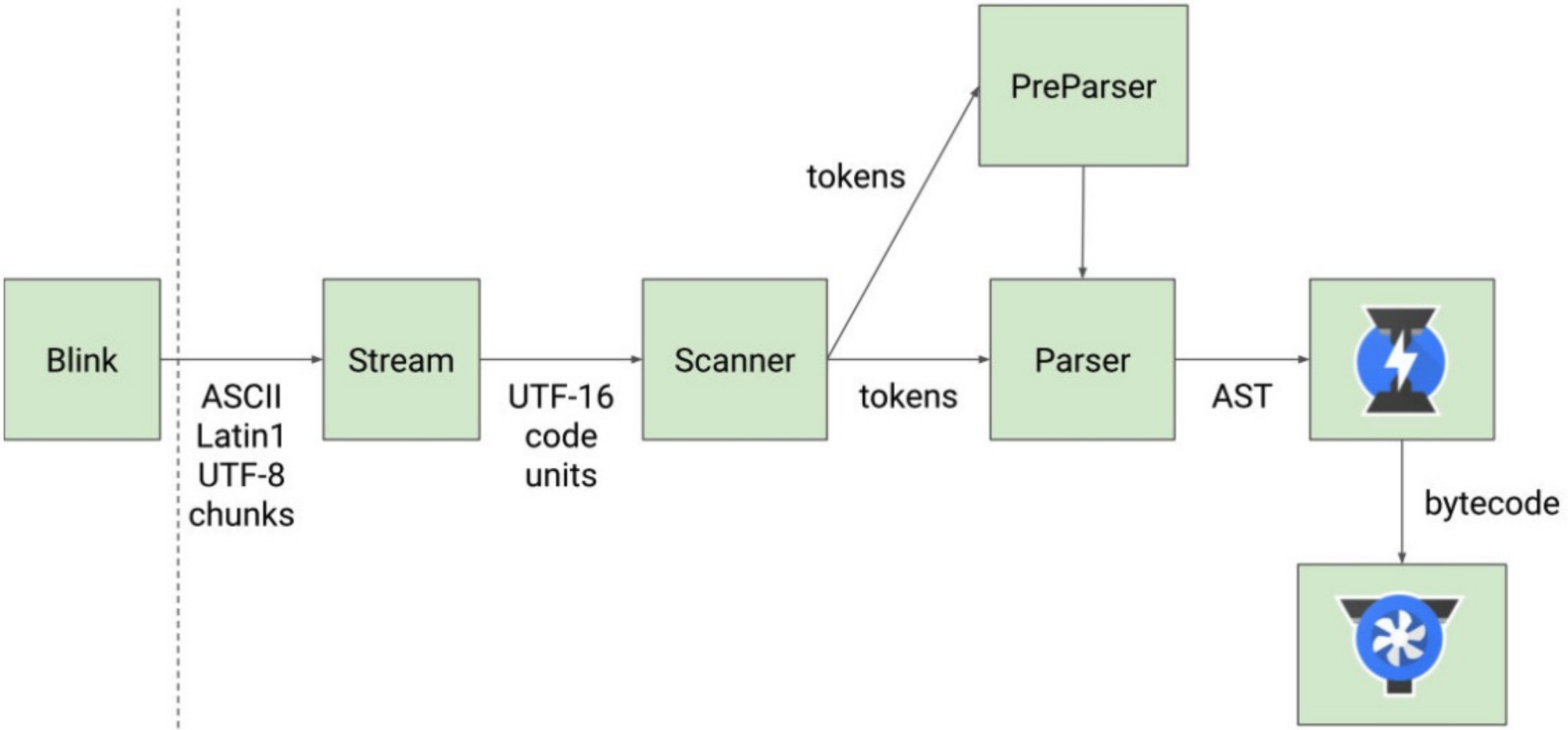
- 我们来看一下官方对V8引擎的定义：
  - V8是用C ++编写的Google开源高性能JavaScript和WebAssembly引擎，它用于Chrome和Node.js等。
  - 它实现ECMAScript和WebAssembly，并在Windows 7或更高版本，macOS 10.12+和使用x64，IA-32，ARM或MIPS处理器的Linux系统上运行。
  - V8可以独立运行，也可以嵌入到任何C ++应用程序中。



V8引擎的架构

- V8引擎本身的源码**非常复杂**，大概有超过**100w行C++代码**，通过了解它的架构，我们可以知道它是如何对JavaScript执行的：
- Parse模块会将JavaScript代码转换成AST（抽象语法树），这是因为解释器并不直接认识JavaScript代码；
  - 如果函数没有被调用，那么是不会被转换成AST的；
  - Parse的V8官方文档：https://v8.dev/blog/scanner
- Ignition是一个解释器，会将AST转换成ByteCode（字节码）
  - 同时会收集TurboFan优化所需要的信息（比如函数参数的类型信息，有了类型才能进行真实的运算）；
  - 如果函数只调用一次，Ignition会执行解释执行ByteCode；
  - Ignition的V8官方文档：https://v8.dev/blog/ignition-interpreter
- TurboFan是一个编译器，可以将字节码编译为CPU可以直接执行的机器码；
  - 如果一个函数被多次调用，那么就会被标记为热点函数，那么就会经过TurboFan转换成优化的机器码，提高代码的执行性能；
  - 但是，机器码实际上也会被还原为ByteCode，这是因为如果后续执行函数的过程中，类型发生了变化（比如sum函数原来执行的是number类型，后来执行变成了string类型），之前优化的机器码并不能正确的处理运算，就会逆向的转换成字节码；
  - TurboFan的V8官方文档：https://v8.dev/blog/turbofan-jit

V8引擎的解析图（官方）


PreParser
tokens
Blink
ASCII
Latin1
UTF-8
chunks
Stream
UTF-16
code
units
Scanner
tokens
Parser
AST
bytecode

V8执行的细节

- **那么我们的JavaScript源码是如何被解析（Parse过程）的呢？**
- Blink将源码交给V8引擎，Stream获取到源码并且进行编码转换；
- Scanner会进行词法分析（lexical analysis），词法分析会将代码转换成tokens；
- 接下来tokens会被转换成AST树，经过Parser和PreParser：
  - Parser就是直接将tokens转成AST树架构；
  - PreParser称之为预解析，为什么需要预解析呢？
    - ✓ 这是因为并不是所有的JavaScript代码，在一开始时就会被执行。那么对所有的JavaScript代码进行解析，必然会影响网页的运行效率；
    - ✓ 所以V8引擎就实现了Lazy Parsing（延迟解析）的方案，它的作用是将不必要的函数进行预解析，也就是只解析暂时需要的内容，而对函数的全量解析是在函数被调用时才会进行；
    - ✓ 比如我们在一个函数outer内部定义了另外一个函数inner，那么inner函数就会进行预解析；
- 生成AST树后，会被Ignition转成字节码（bytecode），之后的过程就是代码的执行过程（后续会详细分析）。

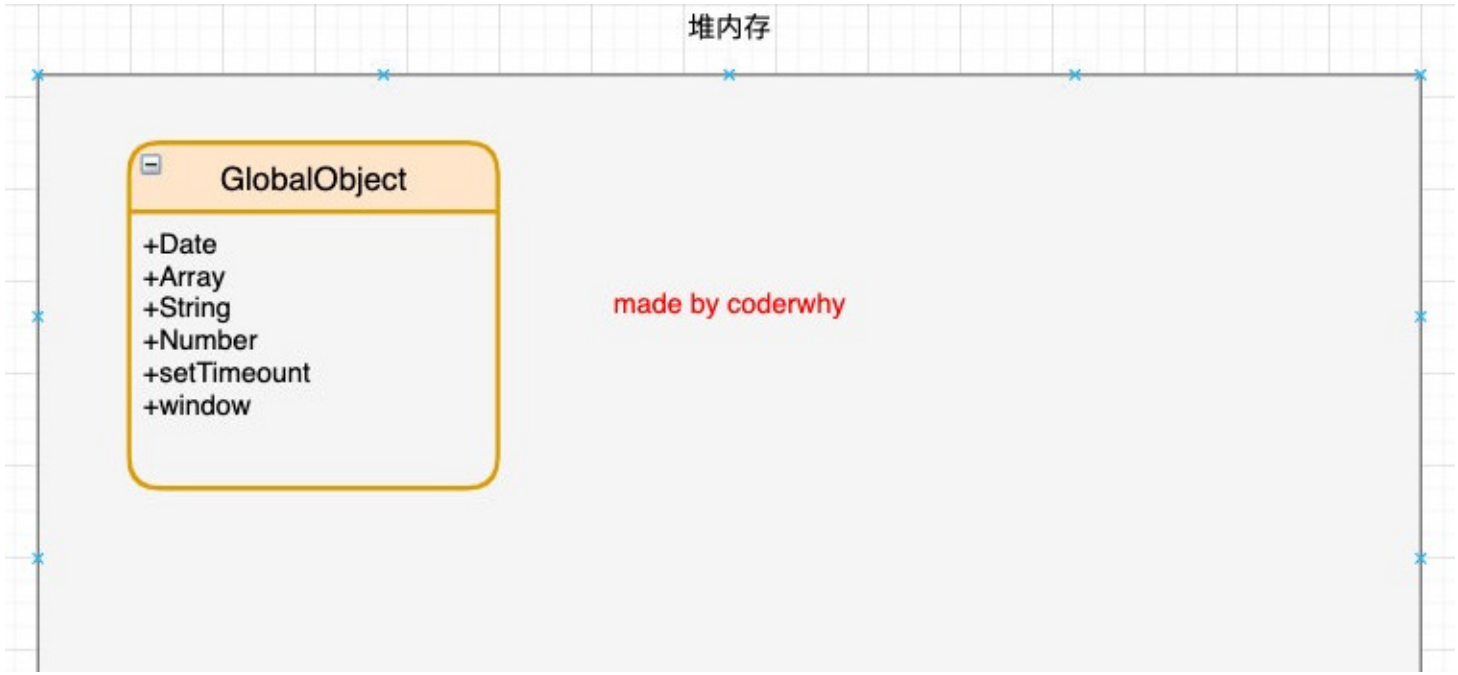
JavaScript的执行过程

- 假如我们有下面一段代码，它在JavaScript中是如何被执行的呢？

```
var name = "why"
function foo() {
  var name = 'foo'
  console.log(name)
}

var num1 = 20
var num2 = 30
var result = num1 + num2

console.log(result)

foo()
```

初始化全局对象

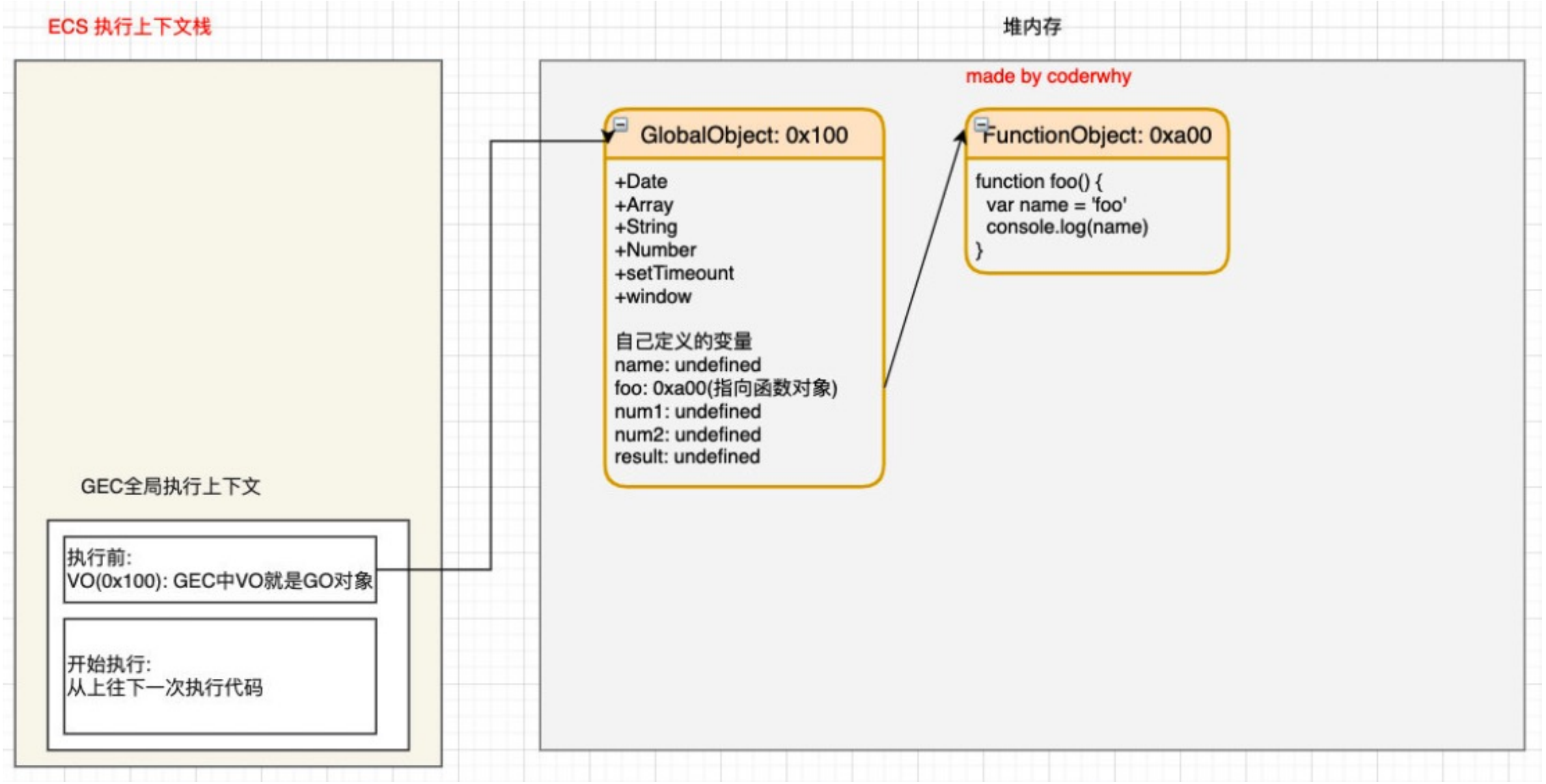
- js引擎会在执行代码之前，会在堆内存中创建一个全局对象：Global Object（GO）
  - 该对象 所有的作用域（scope）都可以访问；
  - 里面会包含Date、Array、String、Number、setTimeout、setInterval等等；
  - 其中还有一个window属性指向自己；



执行上下文栈（调用栈）

- js引擎内部有一个**执行上下文栈（Execution Context Stack，简称ECS）**，它是用于执行**代码的调用栈。**
- 那么现在它要执行谁呢？执行的是**全局的代码块**：
  - 全局的代码块为了执行会构建一个 **Global Execution Context（GEC）**；
  - GEC会 被放入到ECS中 执行；
- **GEC被放入到ECS中里面包含两部分内容：**
  - **第一部分：**在代码执行前，在parser转成AST的过程中，会将全局定义的变量、函数等加入到GlobalObject中，但是并不会赋值；
    - ✓ 这个过程也称之为变量的作用域提升（hoisting）
  - **第二部分：**在代码执行中，对变量赋值，或者执行其他的函数；

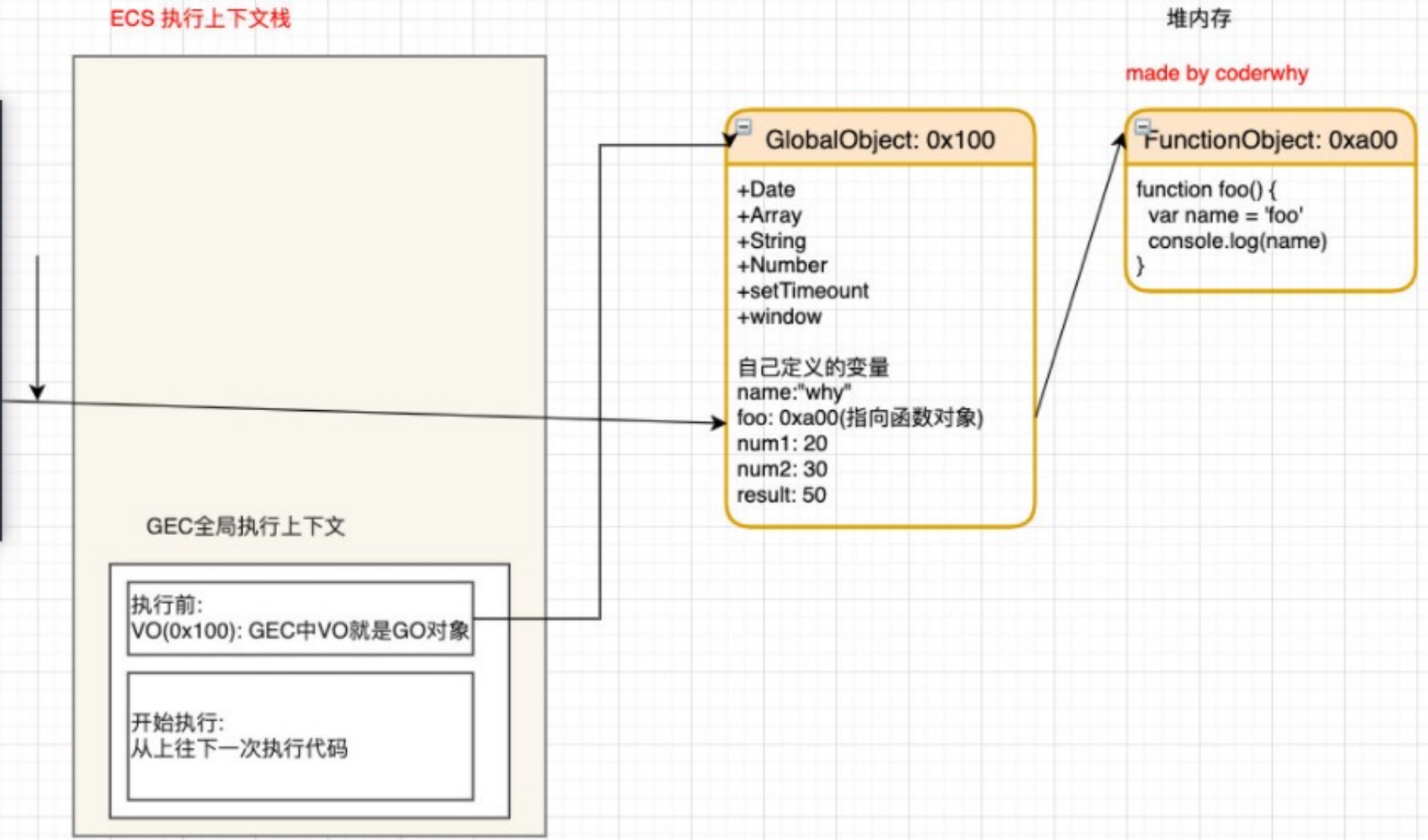
GEC被放入到ECS中



GEC开始执行代码

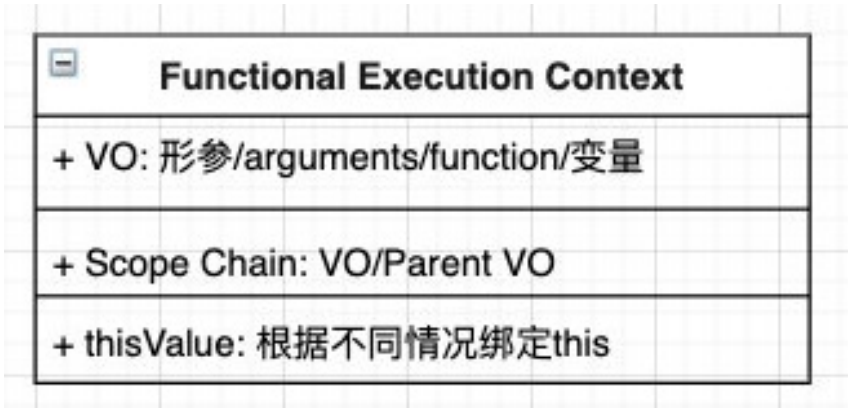
```
var name = "why"
function foo() {
  var name = 'foo'
  console.log(name)
}

var num1 = 20
var num2 = 30
var result = num1 + num2

foo()
```

代码一次执行改变GO

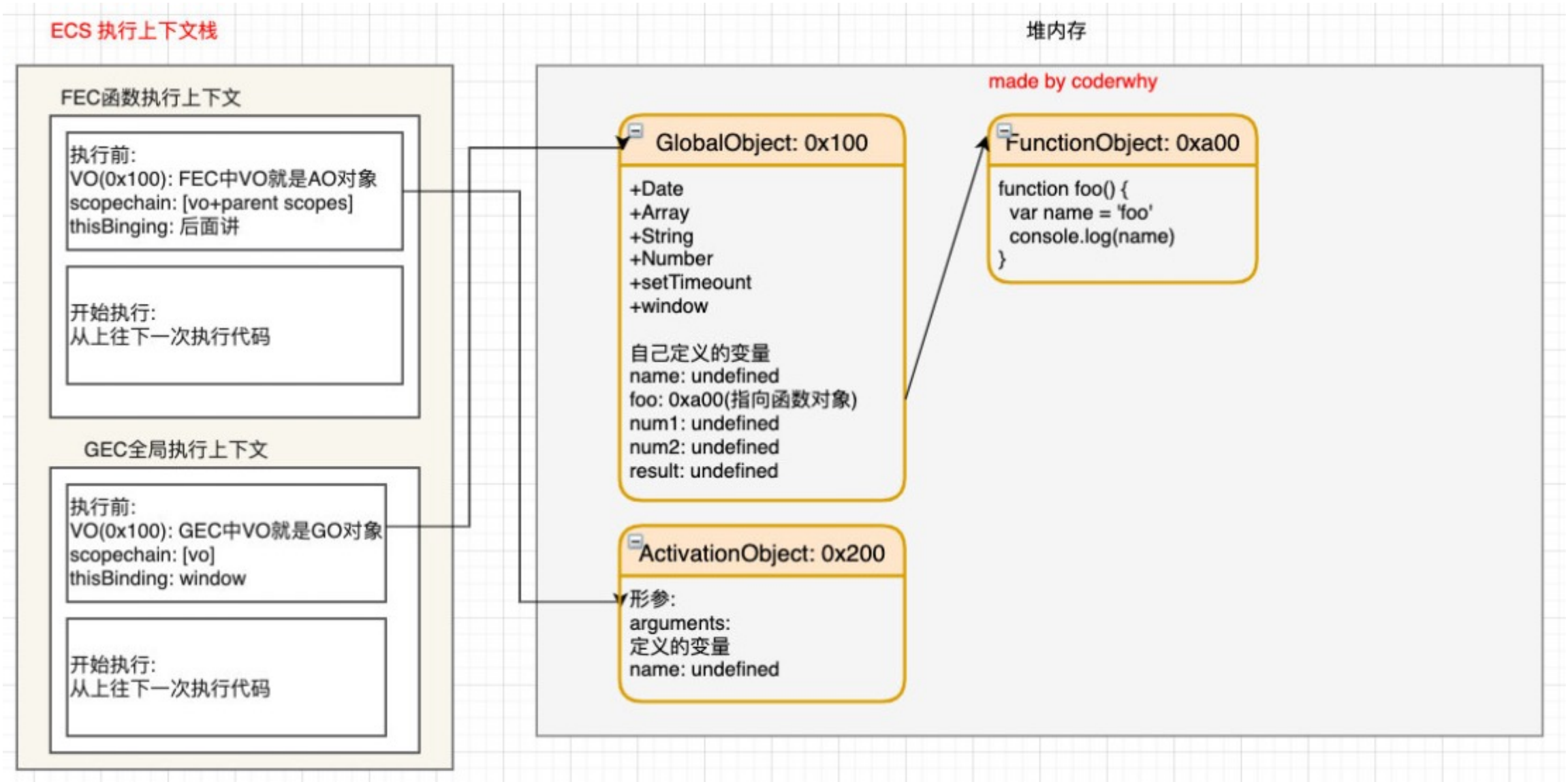


遇到函数如何执行？

- 在执行的过程中**执行到一个函数时**，就会根据**函数体**创建一个**函数执行上下文（Functional Execution Context，简称FEC）**，并且压入到**EC Stack**中。
- FEC中包含三部分内容：
  - 第一部分：在解析函数成为AST树结构时，会创建一个Activation Object（AO）：
    - ✓ AO中包含形参、arguments、函数定义和指向函数对象、定义的变量；
  - 第二部分：作用域链：由VO（在函数中就是AO对象）和父级VO组成，查找时会一层层查找；
  - 第三部分：this绑定的值：这个我们后续会详细解析；

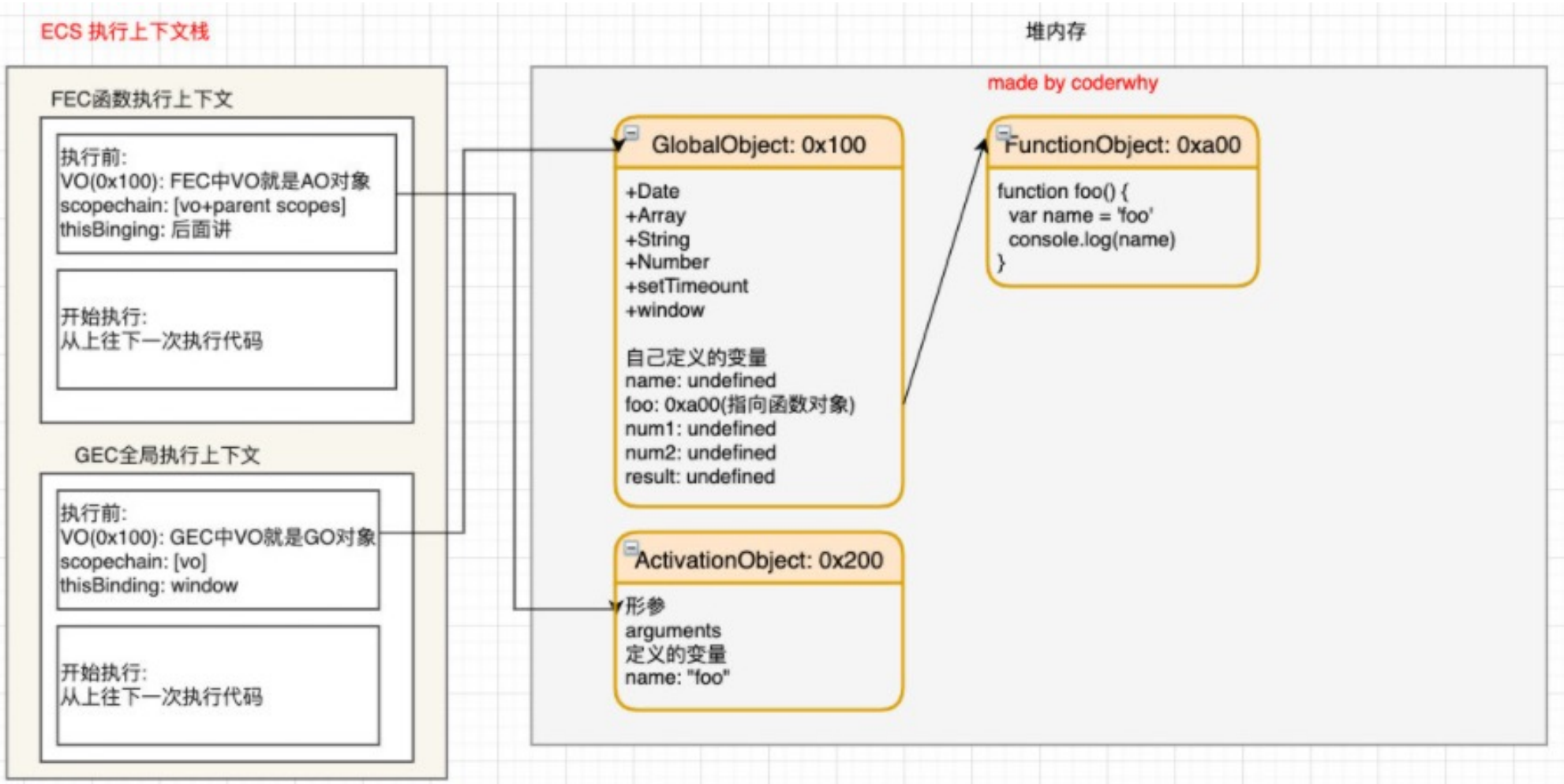


FEC被放入到ECS中



FEC开始执行代码

```
function foo() {
  var name = 'foo'
  console.log(name)
}
```



变量环境和记录

- 其实我们上面的讲解都是基于早期ECMA的版本规范：

> Every execution context has associated with it a variable object. Variables and functions declared in the source text are added as properties of the variable object. For function code, parameters are added as properties of the variable object.
>
> 每一个执行上下文会被关联到一个变量环境（variable object，VO），在源代码中的变量和函数声明会被作为属性添加到VO中。
>
> 对于函数来说，参数也会被添加到VO中。

- 在最新的ECMA的版本规范中，对于一些词汇进行了修改：

> Every execution context has an associated VariableEnvironment. Variables and functions declared in ECMAScript code evaluated in an execution context are added as bindings in that VariableEnvironment's Environment Record. For function code, parameters are also added as bindings to that Environment Record.
>
> 每一个执行上下文会关联到一个变量环境（VariableEnvironment）中，在执行代码中变量和函数的声明会作为环境记录（Environment Record）添加到变量环境中。
>
> 对于函数来说，参数也会被作为环境记录添加到变量环境中。

- 通过上面的变化我们可以知道，在最新的ECMA标准中，我们前面的变量对象VO已经有另外一个称呼了变量环境VE。

作用域提升面试题

200 100 100

200

```
var n = 100
function foo() {
  n = 200
}
foo()

console.log(n)
```

underfind

```
var a = 100

function foo() {
  console.log(a)
  return
  var a = 100
}

foo()
```

underfind 200

```
function foo() {
  console.log(n)
  var n = 200
  console.log(n)
}

var n = 100
foo()
```

=> var 1=100 b=100

```
function foo() {
  var a = b = 100
}

foo()

console.log(a)
console.log(b)
```

a是报错找不到
b打印100

```
var n = 100

function foo1() {
  console.log(n) // 2.100
}

function foo2() {
  var n = 200
  console.log(n) // 1.200
  foo1()
}

foo2()
console.log(n) // 3.100
```

```
function foo(){
    var m = 100
}
foo()
console.log(m) // underfind
```

```
function foo(){
    m = 100
}
foo()
console.log(m)
// 100
```